Scalable XML Collaborative Editing with Undo short paper

Stéphane Martin[1], Pascal Urso[2], and Stéphane Weiss[2]

[1] stephane.martin@lif.univ-mrs.fr
Laboratoire d'Informatique Fondamentale
39 rue F. Jolio-Curie,
13013 Marseille, France
[2] (pascal.urso,stephane.weiss)@loria.fr
Université de Lorraine
LORIA, Campus Scientifique,
54506 Vandoeuvre-lès-Nancy, France

Abstract. Commutative Replicated Data-Type (CRDT) is a new class of algorithms that ensures scalable consistency of replicated data. It has been successfully applied to collaborative editing of texts without complex concurrency control.
In this paper, we present a CRDT to edit XML data. Compared to existing approaches for XML collaborative editing, our approach is more scalable and handles all the XML editing aspects : elements, contents, attributes and undo. Indeed, undo is recognized as an important feature for collaborative editing that allows to overcome system complexity through error recovery or collaborative conflict resolution.

Keywords: XML, Collaborative Editing, P2P, Group Undo, Scalability, Optimistic Replication, CRDT.

1 Introduction

In large-scale infrastructures such as clouds or peer-to-peer networks, data are replicated to ensure availability, efficiency and fault-tolerance. Since data are the heart of the information systems, the consistency of the replicas is a key issue. Mechanisms to ensure strong consistency levels – such as linear or atomic – do not scale, thus modern large-scale infrastructures now rely on eventual consistency.

Commutative Replicated Data Types [12,16] (CRDT) is a promising new class of algorithms used to build operation-based optimistic replication [14] mechanisms. It ensures eventual consistency of replicated data without complex concurrency control. It has been successfully applied to scalable collaborative editing of textual document but not yet on semi-structured data types. EXtensible Markup Language (XML) is used in a wide range of information systems from semi-structured data storing to query. Moreover, XML is the standard format for exchanging data, allowing interoperability and openness.

Collaborative editing (CE) provides several advantages such as obtaining different viewpoints, reducing task completion time, and obtaining a more accurate final result.

Nowadays, collaborative editing becomes massive and part of our every day life. The online encyclopedia Wikipedia users have produced 15 millions of articles in a few years.

Undo has been recognized as an important feature of single and collaborative editors [2,3]. The undo feature provides a powerful way to recover from errors and vandalism acts or to manage edit conflicts. So, it helps the user to face the complexity of the system. However, designing an undo feature is a non-trivial task. First, in collaborative editing, this feature must allow to undo any operation – and not only the last one – from any user. This is called global selective undo (or anyundo). Second, this undo must be correct from the user's point of view. The system must return in a state such as the undone operation has never been performed.

We propose to design an XML CRDT for collaborative editing. This CRDT handles both aspects of XML trees : elements' children and attributes. The order in the list of the elements' children are treated as in linear structure CRDT. Elements' attributes are treated using a last-writer-wins rule. Our undo is obtained by keeping the previous value given to attributes and operations applied on elements, and then counting concurrent undo and redo operations. A garbage collection mechanism is presented to garbage old operations.

2 State of the art

The Operational Transformation (OT) [13] approach is an operation-based replication mechanism. OT relies on a generic integration algorithm and a set of transformation functions specific to the type of replicated data. Some integration mechanism use states vectors – or context vectors [15] in the presence of undo – to detect concurrency between operations; such mechanisms are not adapted to large-scale infrastructures. Ignat et al. [4] propose to couples an integration mechanism that uses anti-entropy, with some specific transformation functions [11] to obtain P2P XML collaboration. However, this proposition replaces deleted elements by tombstones in the edited document to ensure consistency, making the document eventually growing without limits and proposes no undo.

Martin et al. [9] proposes an XML-tree reconciliation mechanism very similar to a CRDT since concurrent operations commute without transformation. However, this approach does not treat XML element's attributes, which require a specific treatment, since they are unique and unordered. Furthermore, it uses state vector that limits its scalability and proposes no undo feature.

In the field of Data Management, some works give attention to XML replication. Some of them [6,1] suppose the existence of some protocol to ensure consistency of replicated content without defining it. Finally, [8] proposes a merging algorithm for concurrent modifications that can only be used in a centralized context.

3 XML CRDT without undo

In a collaborative editor, to ensure scalability and high-responsiveness of local modifications, data must be replicated. This replication is optimistic since local modifications

are immediately executed. The replicas are allowed to diverge in the short time, but the system must ensure eventual consistency. When the system is idle (i.e., all modifications are delivered), the replicas must have the same content.

A Commutative Replicated Data Type(CRDT) [12] is a data type where all concurrent operations commute. In other words, whatever the delivery order of operations, the resulting document is identical. As a result, a CRDT ensure eventual consistency as proven in [12].

Thus, we see an XML collaborative editor as a set of network nodes that host a set of replicas (up to one per node) of the shared XML document. Local modifications are immediately executed and disseminated to all other replicas. We assume that every replica will eventually receive every modification.

We consider an XML tree as an *edge e* with three elements : $e.identifier$ the unique identifier of the edge (a timestamp), $e.children$ the children of the edge (a set of edge), and $e.attributes$ the attributes of the edge (a map string to value). The key of the map are the attribute's name (a string), and a value *av* has two elements, $av.value$: the current value of the attribute (a string), $av.timestamp$: the current timestamp of the attribute. The basic operations that affect an XML tree are :

- $Add(id_p, id)$: Adds a edge with identifier *id* under the edge id_p. This edge is empty, it has no tag-name, child or attribute.
- $Del(id)$: Deletes the edge identified by *id*.
- $SetAttr(id, attr, val, ts)$: Sets the value *val* with the timestamp *ts* to the attribute *attr* of the edge identified by *id*. The deletion of an attribute is done by setting is value to nil.

To allow *Add* and *Del* operations to commute, we use a unique timestamp identifier. Timestamp identifiers can be defined as follows: each replica is identified by a unique identifier *s* and each operation generated by this site is identified by a clock h_s (logical clock or wall clock). An identifier *id* is a pair $(h_s : s)$. For instance $(3 : 2)$ identifies the operation 3 of the site number 2. The set of the identifiers is denoted by *ID*. Thus, two edges added concurrently at the same place in the tree have different identifiers.

To allow *SetAttr* operations to commute, we use a classical last-writer-wins technique. We associate to each attribute a timestamp *ts*. A remote *SetAttr* is applied if and only if its timestamp is higher than the timestamp associated to the attribute. Timestamps are totally ordered. Let $ts_1 = (h_1 : s_1)$ and $ts_2 = (h_2 : s_2)$, we have $ts_1 > ts_2$ if and only if $h_1 > h_2$, or $h_1 = h_2$ and $s_1 > s_2$. Clocks are loosely synchronized, i.e., when a replica receives an operation with a timestamp $(h_2 : s_2)$, it sets its own clock h_1 to $max(h_1, h_2)$.

Special attributes. The special attributes $@tag$ and $@position$ contain the tag-name and the position of an edge and cannot be nil. The position allows to order the children of a node. This position is not a basic number. Indeed, to ensure that the order among edges is the same on all replicas, this position must be *unique, totally ordered and dense*. Positions are dense if a replica can always generate a position between two arbitrary positions. This position can be a priority string concatenated with an identifier [9], a sequence of integers [17], or a bitstring concatenated with an identifier [12] all with a lexicographic ordering. Finally, to model the textual edges we use another special

attribute *@text*. If this attribute has a value *v*, whatever the value of other attributes, the edge is considered as a textual edge with content *v*.

Algorithms The function deliver(op,t) applies an operation op on an XML tree t. The function find(t,id) returns the edge identified by id. The function findFather(t,id) returns the father of the edge identified by id.

```
deliver (Add(id_p, id), t) :
  edge p = find(id_p, t), e = new edge(id);
  if p ≠ nil then p.children = p.children ∪ {e};
end
deliver (Del(id), t) :
  edge p = findFather(id, t), e = find(id, p);
  if p ≠ nil then p.children = p.children \ {e};
end
deliver (SetAttr(id, attr, val, ts), t) :
  edge e = find(id, t);
  if e ≠ nil and (e.attributes[attr] = nil or e.attributes[attr].timestamp < ts) then
      e.attributes[attr].value = val;
      e.attributes[attr].timestamp = ts;
  endif
end
```

4 XML CRDT with undo

Obtaining a correct undo from the user's point of view is a non-trivial task. Let's have the following scenario 1. a user adds an element, 2. a user deletes this element, 3. the add is undone, 4. the delete is undone concurrently by 2 different users. At the end, since both operations add and delete are undone, the node must be invisible. And this must be true on every replica and whatever the delivery order of operations. For instance, using *Del* to undo *Add* leads to different results according to the reception order of the operations. The element is visible if an un-delete is received in last or not if it is a un-add. Such a behavior violates eventual consistency.

To obtain a satisfying undo, we keep the information about every operations applied to each edge. Then we count the *effect counter* of an operation : one minus the number of undo plus the number of redo. If this effect counter is greater than 0, the operation has an effect. An element is visible if the add has an effect counter greater than 0, and no delete with an effect counter greater than 0. The value of an attribute is determined by the more recent value with an effect counter greater than 0. Thus, we need to keep into the map of attributes, the list of values – including nil values – associated to an effect counter. The list is ordered by the decreasing timestamp.

With undo, an edge attribute $e.attributes[attr]$ becomes an ordered list of *value v*, each value containing 3 elements : *v.value* a value of the attribute (a string), *v.timestamp* the timestamp associated to this value, and $v.effect$ the effect counter of this value (a integer). *The list is ordered by the timestamp*. The function add(l,v) adds a value v in

the list l at its place according to $v.timestamp$. The function get(l, ts) returns the value associated to ts in the list l. The special $@add$ attribute has only one value associated to the timestamp equal to the edge identifier. The special $@del$ attribute stores the list of timestamp of delete operations applied to the edge.

```
deliver (Add(id_p, id), t) :
  edge p = find(t, id_p), e = new edge(id);
  p.children = p.children ∪ {e}
  add(e.attributes[@add], new value (nil, id, 1));
end
deliver (Del(id, ts), t) :
  edge e = find(t, id);
  add(e.attributes[@del], new value (nil, ts, 1));
end
deliver (SetAttr(id, attr, val, ts), t) :
  edge e = find(t, id);
  add(e.attributes[attr], new value (val, ts, 1));
end
```

Undo of an operation is simply achieved by decrementing the corresponding effect counter. When a *Redo* is delivered, the increment function is called with a delta of $+1$.

```
deliver (Undo(Add(id_p, id)), t) :
  increment(t, id, @add, id, −1);
end
deliver (Undo(Del(id, ts)), t) :
  increment(t, id, @del, ts, −1);
end
deliver (Undo(SetAttr(id, attr, val, ts)), t) :
  increment(t, id, attr, ts, −1);
end
function increment(t, id, attr, ts, delta)
  edge e = find(t, id);
  value v = get(e.attributes[attr], ts);
  v. effect  += delta;
end
```

Exemple. Figure 1 presents the application of our functions on the introducing example. On every replica, the add and del operations have an effect counter lesser or equal to 0. Thus none of these operations have an effect on the XML tree and the edge is invisible.

Model to XML. As the model described above includes tombstones and operations information, it cannot be used directly by applications. Indeed, applications must not see a tombstones and only one value for each attribute. A node is visible if the effect counter of the attribute $@add$ is at least one, and if all values of the attribute $@del$ have an effect counter of at most 0. If the edge is a text, i.e., the attribute $@text$ has a

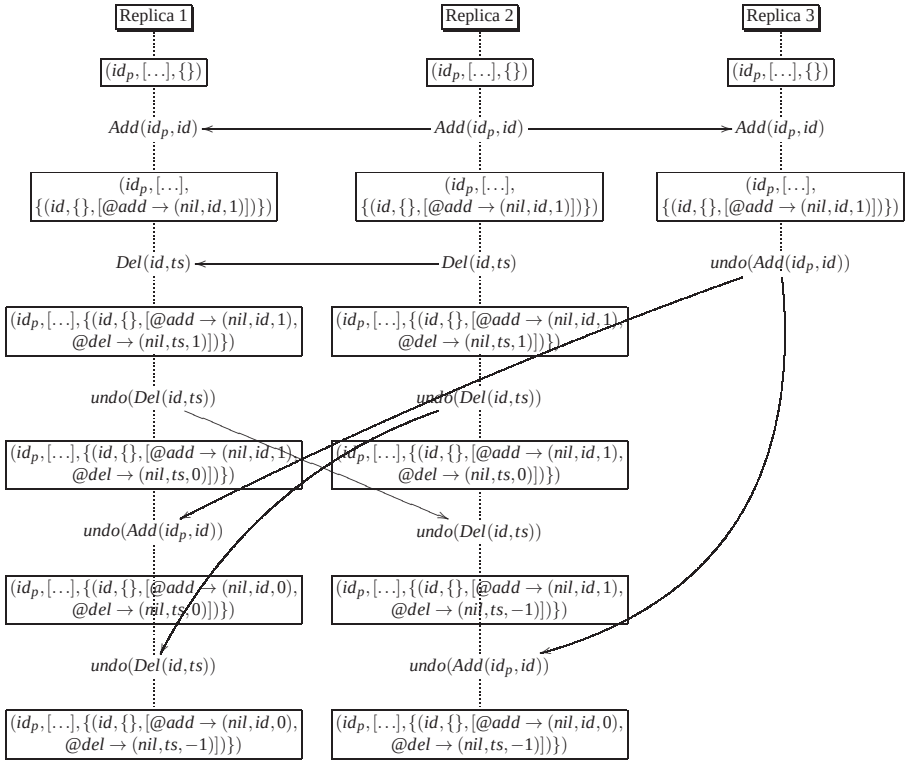

Fig. 1. Concurrent undos with effect counters.

value, we write this value in the XML Document. If the edge is visible and is not a text, we write the tag and the attributes corresponding to that edge. Therefore, we need to compute the current value of the attributes which is the newest non-undone value of a value list. Finally, the rendering function calls itself to treat the children of the edge.

Correctness. To ensure that our data type is a CRDT and thus that eventual consistency is ensured, we must prove all our operations commutes. For a complete proof, please see [10]. The only requirement to ensure consistency of the XML CRDT without undo is to receive delete operation after insert of a node. With undo, this constraint is not required to ensure consistency since a delete can be received before an insert. The delete produces directly a tombstone.

5 Garbage collecting

Concerning the scalability in term of operations number, the XML CRDT without undo requires tombstones for attributes as the Thomas Write Rule defined in the RFC 677 [5]. The XML CRDT with undo requires to keep an information about every operation applied to the XML document. This is not surprising since any undo system must keep

a trace of an operation that can be undone, either in the document model or in a history log.

However, a garbage collecting mechanism can be designed. Such a garbage collection is similar to the one already present in the RFC 667 [5]. Each replica i maintains a vector v_i of the last clock timestamp received by all other replicas (including its own clock). From this vector the replica computes m_i the minimum of these clocks. This minimum is sent regularly to the other replicas. It can be piggybacked to operation's messages or sent regularly in a specific message. From the minimum received (including its own), each replica maintains another vector V_i. The minimum of V_i is M_i. The point is that, if communication is FIFO, a replica knows that every replica has received all potential messages with a timestamp less or equal to M_i. Thus any tombstone with a timestamp less or equal to M_i can be safely removed. This mechanism can be directly used in the XML CRDT without undo to remove old deleted attributes.

In the XML CRDT with undo, we only authorize to produce an undo of an operation whose timestamp is greater than m_i. Thus operations with a timestamp lesser than M_i will never see their effect modified. So, elements such as follows can be safely and definitively purged :

- attribute value v with $v.timestamp < M_i$ and $v.effect \leq 0$
- attribute value v with $v.timestamp < M_i$ and there exists v' with $v.timestamp < v'.timestamp < M_i$ and $v'.effect > 0$
- attribute with no value or with every value v such that $v.timestamp < M_i$ and ($v.effect \leq 0$ or $v.value = nil$)
- edge with any delete value d with $d.timestamp < M_i$ and $d.effect > 0$ or with the add value a with $a.timestamp < M_i$ and $a.effect \leq 0$.

Thus, the time and space complexity of the approach is greatly reduced to be proportional to the size of the view. Moreover, differently to the RFC 677, replicas send $m_i - k$ with k a global constant instead of m_i. Thus, even if the replicas are tightly synchronized – having m_i very close to their own clock –, the replicas can always undo the last operations. Also, the garbage collecting mechanism that can be adapted to the other tombstone-based approach is much less scalable since based on a consensus-like method [7].

6 Conclusion

We have presented a commutative replicated data type that supports XML collaborative editing, including a global selective undo mechanism. Our commutative replicated data type is designed to scale since the replicas number never impacts the execution complexity. Obviously, the undo mechanism requires to keep information about the operations we allow to undo. We presented a garbage collection mechanism that allows to purge the old operations information.

We still have much work to achieve on this topic. Firstly, we need to make experiments to establish the actual scalability and efficiency of the approach in presence of huge data. Secondly, we plan to study replication of XML data typed with DTD or XSD. This is a difficult task, never achieved in a scalable way.

References

1. S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo. Dynamic xml documents with distribution and replication. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 527–538, New York, NY, USA, 2003. ACM.
2. G. D. Abowd and A. J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.
3. R. Choudhary and P. Dewan. A general multi-user undo/redo model. In *ECSCW'95: Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work*, pages 231–246, Norwell, MA, USA, 1995. Kluwer Academic Publishers.
4. C.-L. Ignat and G. Oster. Peer-to-peer collaboration over xml documents. In Y. Luo, editor, *CDVE*, volume 5220 of *Lecture Notes in Computer Science*, pages 66–73. Springer, 2008.
5. P. R. Johnson and R. H. Thomas. RFC 677: Maintenance of duplicate databases, January 1975, (Septembre 2005). `http://www.ietf.org/rfc/rfc677.txt`.
6. G. Koloniari and E. Pitoura. Peer-to-peer management of xml data: issues and research challenges. *SIGMOD Rec.*, 34(2):6–17, 2005.
7. M. Letia, N. Preguiça, and M. Shapiro. CRDTs: Consistency without concurrency control. In *SOSP W. on Large Scale Distributed Systems and Middleware (LADIS)*, pages 29–34, Big Sky, MT, USA, October 2009. sigops, acm.
8. T. Lindholm. Xml three-way merge as a reconciliation engine for mobile data. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 93–97, New York, NY, USA, 2003. ACM.
9. S. Martin and D. Lugiez. Collaborative peer to peer edition: Avoiding conflicts is better than solving conflicts. In H. Weghorn and P. T. Isaías, editors, *IADIS AC (2)*, pages 124–128. IADIS Press, 2009.
10. S. Martin, P. Urso, and S. Weiss. Scalable XML Collaborative Editing with Undo. Research Report RR-7362, INRIA, 08 2010.
11. G. Oster, P. Urso, P. Molli, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *The Second International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2006)*, Atlanta, Georgia, USA, November 2006. IEEE Press.
12. N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *ICDCS*, pages 395–403. IEEE Computer Society, 2009.
13. M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW*, pages 288–297, 1996.
14. Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.
15. D. Sun and C. Sun. Operation Context and Context-based Operational Transformation. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 279–288, Banff, Alberta, Canada, November 2006. ACM Press.
16. S. Weiss, P. Urso, and P. Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *ICDCS*, pages 404 –412, Montréal, Québec, Canada, jun. 2009. IEEE Computer Society.
17. S. Weiss, P. Urso, and P. Molli. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2009.